Exhibit JX19

- *Internal messages with destinations in this block* — The reason for their inclusion is their presence in *OutMsgQueue* of the most recent state of a neighboring shardchain,[26] or their presence in *OutMsgDescr* of this very block. This neighboring shardchain is completely determined by the transit address indicated in the forwarded message envelope, which is replicated in *InMsg* as well. The "fate" of this message is again described by a reference to the processing transaction inside the current block.

- *Immediately routed internal messages* — Essentially a subclass of the previous class of messages. In this case, the imported message is one of the outbound messages generated in this very block.

- *Transit internal messages* — Have the same reason for inclusion as the previous class of messages. However, they are not processed inside the block, but internally forwarded into *OutMsgDescr* and *OutMsgQueue*. This fact, along with a reference to the new envelope of the transit message, must be registered in *InMsg*.

- *Discarded internal messages with destinations in this block* — An internal message with a destination in this block may be imported and immediately discarded instead of being processed by a transaction if it has already been received and processed via IHR in a preceding block of this shardchain. In this case, a reference to the previous processing transaction must be provided.

- *Discarded transit internal messages* — Similarly, a transit message may be discarded immediately after import if it has already been delivered via IHR to its final destination. In this case, a Merkle proof of its processing in the final block (as an IHR message) is required.

3.2.2. Descriptor of an inbound message. Each inbound message is described by an instance of the `InMsg` type, which has six constructors corresponding to the cases listed above in **3.2.1**:

```
msg_import_ext$000 msg:^Message transaction:^Transaction
              = InMsg;
msg_import_ihr$010 msg:^Message transaction:^Transaction
```

[26] Recall that a shardchain is considered a neighbor of itself.

```
    ihr_fee:Grams proof_created:^Cell = InMsg;
msg_import_imm$011 in_msg:^MsgEnvelope
    transaction:^Transaction fwd_fee:Grams = InMsg;
msg_import_fin$100 in_msg:^MsgEnvelope
    transaction:^Transaction fwd_fee:Grams = InMsg;
msg_import_tr$101  in_msg:^MsgEnvelope out_msg:^MsgEnvelope
    transit_fee:Grams = InMsg;
msg_discard_fin$110 in_msg:^MsgEnvelope transaction_id:uint64
    fwd_fee:Grams = InMsg;
msg_discard_tr$111 in_msg:^MsgEnvelope transaction_id:uint64
    fwd_fee:Grams proof_delivered:^Cell = InMsg;
```

Notice that the processing transaction is referred to in the first four constructors directly by a cell reference to `Transaction`, even though the logical time of the transaction `transaction_lt:uint64` would suffice for this purpose. Internal consistency conditions ensure that the transaction referred to does belong to the destination smart contract indicated in the message, and that the inbound message processed by that transaction is indeed the one being described in this *InMsg* instance.

Furthermore, notice that `msg_import_imm` could be distinguished from `msg_import_fin` by observing that it is the only case when the logical creation time of the message being processed is greater than or equal to the (minimal) logical time of the block importing the message.

3.2.3. Collecting forwarding and transit fees from imported messages. The *InMsg* structure is also used to indicate the forwarding and transit fees collected from inbound messages. The fee itself is indicated in `ihr_fee`, `fwd_fee`, or `transit_fee` fields; it is absent only in inbound external messages, which use other mechanisms to reward the validators for importing them. The fees must satisfy the following internal consistency conditions:

- For external messages (`msg_import_ext`), there is no forwarding fee.
- For IHR-imported internal messages (`msg_import_ihr`), the fee equals `ihr_fee`, which must coincide with the `ihr_fee` value indicated in the message itself. Notice that `fwd_fee` or `fwd_fee_remaining` are never collected from IHR-imported messages.

- For internal messages delivered to their destination (`msg_import_fin` and `msg_import_imm`), the fee equals the `fwd_fee_remaining` of the enveloped inbound message `in_msg`. Note that it cannot exceed the `fwd_fee` value indicated in the message itself.
- For transit messages (`msg_import_tr`), the fee equals the difference between the `fwd_fee_remaining` values indicated in the `in_msg` and `out_msg` envelopes.
- For discarded messages, the fee also equals the `fwd_fee_remaining` indicated in `in_msg`.

3.2.4. Imported value of an inbound message. Each imported message imports some value—a certain amount of one or more cryptocurrencies—into the block. This imported value is computed as follows:

- An external message imports no value.
- An IHR-imported message imports its `value` plus its `ihr_fee`.
- A delivered or transit internal message imports its `value` plus its `ihr_fee` plus the value of `fwd_fee_remaining` of its `in_msg` envelope.
- A discarded message imports the `fwd_fee_remaining` of its `in_msg` envelope.

Notice that the forwarding and transit fees collected from an imported message do not exceed its imported value.

3.2.5. Augmented hashmaps, or dictionaries. Before continuing, let us discuss the serialization of *augmented* hashmaps, or dictionaries.

Augmented hashmaps are key-value storage structures with n-bit keys and values of some type X, similar to the ordinary hashmaps described in [4, 3.3]. However, each intermediate node of the Patricia tree representing an augmented hashmap is *augmented* by a value of type Y.

These augmentation values must satisfy certain *aggregation conditions*. Typically, Y is an integer type, and the aggregation condition is that the augmentation value of a fork must equal the sum of the augmentation values of its two children. In general, a *fork evaluation function* $S : Y \times Y \to Y$ or $S : Y \to Y \to Y$ is used instead of the sum. The augmentation value of a leaf is usually computed from the value stored in that leaf by means of a *leaf*

evaluation function $L : X \to Y$. The augmentation value of a leaf may be stored explicitly in the leaf along with the value; however, in most cases there is no need for this, because the leaf evaluation function L is very simple.

3.2.6. Serialization of augmented hashmaps. The serialization of augmented hashmaps with n-bit keys, values of type X, and augmentation values of type Y is given by the following TL-B scheme, which is an extension of the one provided in [4, 3.3.3]:

```
ahm_edge#_ {n:#} {X:Type} {Y:Type} {l:#} {m:#}
  label:(HmLabel ~l n) {n = (~m) + l}
  node:(HashmapAugNode m X Y) = HashmapAug n X Y;
ahmn_leaf#_ {X:Type} extra:Y value:X = HashmapAugNode 0 X Y;
ahmn_fork#_ {n:#} {X:Type} extra:Y left:^(HashmapAug n X Y)
  right:^(HashmapAug n X Y) = HashmapAugNode (n+1) X Y;

ahme_empty$0 {n:#} {X:Type} {Y:Type} extra:Y
          = HashmapAugE n X Y;
ahme_root$1 {n:#} {X:Type} {Y:Type} extra:Y
  root:^(HashmapAug n X Y) = HashmapAugE n X Y;
```

3.2.7. Augmentation of *InMsgDescr*. The collection of inbound message descriptors is augmented by a vector of two currency values, representing the imported value and the forwarding and transit fees collected from a message or a collection of messages:

```
import_fees$_ fees_collected:Grams
  value_imported:CurrencyCollection = ImportFees;
```

3.2.8. Structure of *InMsgDescr*. Now the *InMsgDescr* itself is defined as an augmented hashmap, with 256-bit keys (equal to the representation hashes of imported messages), values of type `InMsg` (cf. **3.2.2**), and augmentation values of type `ImportFees` (cf. **3.2.7**):

```
_ (HashmapAugE 256 InMsg ImportFees) = InMsgDescr;
```

This TL-B notation uses an anonymous constructor `_` to define `InMsgDescr` as a synonym for another type.

3.2.9. Aggregation rules for *InMsgDescr*. The fork evaluation and leaf evaluation functions (cf. **3.2.5**) are not included explicitly in the above notation, because the dependent types of TL-B are not expressive enough for this purpose. In words, the fork evaluation function is just the componentwise addition of two `ImportFees` instances, and the leaf evaluation function is defined by the rules listed in **3.2.3** and **3.2.4**. In this way, the root of the Patricia tree representing an instance of *InMsgDescr* contains an *ImportFees* instance with the total value imported by all inbound messages, and with the total forwarding fees collected from them.

3.3 Outbound message queue and descriptors

This section discusses *OutMsgDescr*, the structure representing all outbound messages of a block, along with their envelopes and brief descriptions of the reasons for including them into *OutMsgDescr*. This structure also describes all modifications of *OutMsgQueue*, which is a part of the shardchain state.

3.3.1. Types of outbound messages. Outbound messages may be classified as follows:

- *External outbound messages*, or "messages to nowhere" — Generated by a transaction inside this block. The reason for including such a message into *OutMsgDescr* is simply a reference to its generating transaction.
- *Immediately processed internal outbound messages* — Generated and processed in this very block, and not included into *OutMsgQueue*. The reason for including such a message is a reference to its generating transaction, and its "fate" is described by a reference to the corresponding entry in *InMsgDescr*.
- *Ordinary (internal) outbound messages* — Generated in this block and included into *OutMsgQueue*.
- *Transit (internal) outbound messages* — Imported into the *InMsgDescr* of the same block and routed via HR until a next-hop address outside the current shard is obtained.

3.3.2. Message dequeueing records. Apart from the above types of outbound messages, *OutMsgDescr* can contain special "message dequeueing

records", which indicate that a message has been removed from the *OutMsgQueue* in this block. The reason for this removal is indicated in the message deletion record; it consists of a reference to the enveloped message being deleted, and of the logical time of the neighboring shardchain block that has this enveloped message in its *InMsgQueue*.

Notice that on some occasions a message may be imported from the *OutMsgQueue* of the current shardchain, internally routed, and then included into *OutMsgDescr* and *OutMsgQueue* again with a different envelope.[27] In this case, a variant of the transit outbound message description is used, which doubles as a message dequeueing record.

3.3.3. Descriptor of an outbound message. Each outbound message is described by an instance of *OutMsg*:

```
msg_export_ext$000 msg:^Message
    transaction:^Transaction = OutMsg;
msg_export_imm$010 out_msg:^MsgEnvelope
    transaction:^Transaction reimport:^InMsg = OutMsg;
msg_export_new$001 out_msg:^MsgEnvelope
    transaction:^Transaction = OutMsg;
msg_export_tr$011  out_msg:^MsgEnvelope
    imported:^InMsg = OutMsg;
msg_export_deq$110 out_msg:^MsgEnvelope
    import_block_lt:uint64 = OutMsg;
msg_export_tr_req$111 out_msg:^MsgEnvelope
    imported:^InMsg = OutMsg;
```

The last two descriptions have the effect of removing (dequeueing) the message from *OutMsgQueue* instead of inserting it. The last one re-inserts the message into *OutMsgQueue* with a new envelope after performing the internal routing (cf. **2.1.11**).

3.3.4. Exported value of an outbound message. Each outbound message described by an *OutMsg* exports some value—a certain amount of one or more cryptocurrencies—from the block. This exported value is computed as follows:

[27] This situation is rare and occurs only after shardchain merge events. Normally the messages imported from the *OutMsgQueue* of the same shardchain have destinations inside this shardchain, and are processed accordingly instead of being re-queued.

- An external outbound message exports no value.
- An internal message, generated in this block, exports its `value` plus its `ihr_fee` plus its `fwd_fee`. Notice that `fwd_fee` must be equal to the `fwd_fee_remaining` indicated in the `out_msg` envelope.
- A transit message exports its `value` plus its `ihr_fee` plus the value of `fwd_fee_remaining` of its `out_msg` envelope.
- The same holds for `msg_export_tr_req`, the constructor of *OutMsg* used for re-inserted dequeued messages.
- A message dequeueing record (`msg_export_deq`; cf. **3.3.2**) exports no value.

3.3.5. Structure of *OutMsgDescr*. The *OutMsgDescr* itself is simply an augmented hashmap (cf. **3.2.5**), with 256-bit keys (equal to the representation hash of the message), values of type *OutMsg*, and augmentation values of type *CurrencyCollection*:

```
_ (HashmapAugE 256 OutMsg CurrencyCollection) = OutMsgDescr;
```

The augmentation is the *exported value* of the corresponding message, aggregated by means of the sum, and computed at the leaves as explained in **3.3.4**. In this way, the total exported value appears near the root of the Patricia tree representing *OutMsgDescr*.

The most important consistency condition for *OutMsgDescr* is that its entry with key k must be an *OutMsg* describing a message m with representation hash $\text{HASH}^\flat(m) = k$.

3.3.6. Structure of *OutMsgQueue*. Recall (cf. **1.2.7**) that *OutMsgQueue* is a part of the blockchain state, not of a block. Therefore, a block contains only hash references to its initial and final state, and its newly-created cells.

The structure of *OutMsgQueue* is simple: it is just an augmented hashmap with 352-bit keys and values of type *OutMsg*:

```
_ (HashmapAugE 352 OutMsg uint64) = OutMsgQueue;
```

The key used for an outbound message m is the concatenation of its 32-bit next-hop *workchain_id*, the first 64 bits of the next-hop address inside that workchain, and the representation hash $\text{HASH}^\flat(m)$ of the message m itself.

The augmentation is by the logical creation time $\text{Lt}(m)$ of message m at the leaves, and by the minimum of the augmentation values of the children at the forks.

The most important consistency condition for *OutMsgQueue* is that the value at key k must indeed contain an enveloped message with the expected next-hop address and representation hash.

3.3.7. Consistency conditions for *OutMsg*. Several internal consistency conditions are imposed on *OutMsg* instances present in *OutMsgDescr*. They include the following:

- Each of the first three constructors of outbound message descriptions includes a reference to the generating transaction. This transaction must belong to the source account of the message, it must contain a reference to the specified message as one of its outbound messages, and it must be recoverable by looking it up by its `account_id` and `transaction_id`.
- `msg_export_tr` and `msg_export_tr_req` must refer to an *InMsg* instance describing the same message (in a different original envelope).
- If one of the first four constructors is used, the message must be absent in the initial *OutMsgQueue* of the block; otherwise, it must be present.
- If `msg_export_deq` is used, the message must be absent in the final *OutMsgQueue* of the block; otherwise, it must be present.
- If a message is not mentioned in *OutMsgDescr*, it must be the same in the initial and final *OutMsgQueue*s of the block.

4 Accounts and transactions

This chapter discusses the layout of *accounts* (or *smart contracts*) and their *state* in the TON Blockchain. It also considers *transactions*, which are the only way to modify the state of an account, and to process inbound messages and generate new outbound messages.

4.1 Accounts and their states

Recall that a *smart contract* and an *account* are the same thing in the context of the TON Blockchain, and that these terms can be used interchangeably, at least as long as only small (or "usual") smart contracts are considered. A *large* smart contract may employ several accounts lying in different shardchains of the same workchain for load balancing purposes.

An account is *identified* by its full address, and is *completely described* by its state. In other words, there is nothing else in an account apart from its address and state.

4.1.1. Account addresses. In general, an account is completely identified by its *full address*, consisting of a 32-bit *workchain_id*, and the (usually 256-bit) *internal address* or *account identifier account_id* inside the chosen workchain. In the basic workchain ($workchain\_id = 0$) and in the masterchain ($workchain\_id = -1$) the internal address is always 256-bit. In these workchains,[28] *account_id* cannot be chosen arbitrarily, but must be equal to the hash of the initial code and data of the smart contract; otherwise, it will be impossible to initialize the account with the intended code and data (cf. **1.7.3**), and to do anything with the accumulated funds in the account balance.

4.1.2. Zero account. By convention, the *zero account* or *account with zero address* accumulates the processing, forwarding, and transit fees, as well as any other payments collected by the validators of the masterchain or a workchain. Furthermore, the zero account is a "large smart contract", meaning that each shardchain has its instance of the zero account, with the most significant bits of the address adjusted to lie in the shard. Any funds transferred to the zero account, intentionally or by accident, are effectively

[28] For simplicity, we sometimes treat the masterchain as just another workchain with $workchain\_id = -1$.

a gift for the validators. For example, a smart contract might destroy itself by sending all its funds to the zero account.

4.1.3. Small and large smart contracts. By default, smart contracts are "small", meaning that they have one account address belonging to exactly one shardchain at any given moment of time. However, one can create a "large smart contract of splitting depth d", meaning that up to 2^d instances of the smart contract may be created, with the first d bits of the original address of the smart contract replaced by arbitrary bit sequences.[29] One can send messages to such smart contracts using internal anycast addresses with `anycast` set to d (cf. **3.1.2**). Furthermore, the instances of the large smart contract are allowed to use this anycast address as the source address of their generated messages.

An instance of a large smart contract is an account with non-zero *maximal splitting depth* d.

4.1.4. The three kinds of accounts. There are three kinds of accounts:

- *Uninitialized* — The account only has a balance; its code and data have not yet been initialized.
- *Active* — The account's code and data have been initialized as well.
- *Frozen* — The account's code and data have been replaced by a hash, but the balance is still stored explicitly. The balance of a frozen account may effectively become negative, reflecting due storage payments.

4.1.5. Storage profile of an account. The *storage profile* of an account is a data structure describing the amount of persistent blockchain state storage used by that account. It describes the total amount of cells, data bits, and internal and external cell references used.

```
storage_used$_ cells:(VarUInteger 7) bits:(VarUInteger 7)
  ext_refs:(VarUInteger 7) int_refs:(VarUInteger 7)
  public_cells:(VarUInteger 7) = StorageUsed;
```

[29] In fact, up to the first d bits are replaced in such a way that each shard contains at most one instance of the large smart contract, and that shards (w, s) with prefix s of length $|s| \leq d$ contain exactly one instance.

The same type `StorageUsed` may represent the storage profile of a message, as required, for instance, to compute `fwd_fee`, the total forwarding fee for Hypercube Routing. The storage profile of an account has some additional fields indicating the last time when the storage fees were exacted:

```
storage_info$_ used:StorageUsed last_paid:uint32
               due_payment:(Maybe Grams) = StorageInfo;
```

The `last_paid` field contains either the unixtime of the most recent storage payment collected (usually this is the unixtime of the most recent transaction), or the unixtime when the account was created (again, by a transaction). The `due_payment` field, if present, accumulates the storage payments that could not be exacted from the balance of the account, represented by a strictly positive amount of nanograms; it can be present only for uninitialized or frozen accounts that have a balance of zero Grams (but may have non-zero balances in other cryptocurrencies). When `due_payment` becomes larger than the value of a configurable parameter of the blockchain, the account is destroyed altogether, and its balance, if any, is transferred to the zero account.

4.1.6. Account description. The state of an account is represented by an instance of type *Account*, described by the following TL-B scheme:[30]

```
account_none$0 = Account;
account$1 addr:MsgAddressInt storage_stat:StorageInfo
          storage:AccountStorage = Account;

account_storage$_ last_trans_lt:uint64
    balance:CurrencyCollection state:AccountState
  = AccountStorage;

account_uninit$00 = AccountState;
account_active$1 _:StateInit = AccountState;
account_frozen$01 state_hash:uint256 = AccountState;

acc_state_uninit$00 = AccountStatus;
acc_state_frozen$01 = AccountStatus;
```

[30]This scheme uses anonymous constructors and anonymous fields, both represented by an underscore `_`.

```
acc_state_active$10 = AccountStatus;
acc_state_nonexist$11 = AccountStatus;

tick_tock$_ tick:Boolean tock:Boolean = TickTock;

_ split_depth:(Maybe (## 5)) special:(Maybe TickTock)
  code:(Maybe ^Cell) data:(Maybe ^Cell)
  library:(Maybe ^Cell) = StateInit;
```

Notice that `account_frozen` contains the representation hash of an instance of *StateInit*, instead of that instance itself, which would otherwise be contained in an `account_active`; `account_uninit` is similar to `account_frozen`, but it does not contain an explicit `state_hash`, because it is assumed to be equal to the internal address of the account (*account_id*), already present in the `addr` field. The `split_depth` field is present and non-zero only in instances of large smart contracts. The `special` field may be present only in the masterchain—and within the masterchain, only in some *fundamental* smart contracts required for the whole system to function.

The storage statistics kept in `storage_stat` reflect the total storage usage of cell slice `storage`. In particular, the bits and cells used to store the `balance` are also reflected in `storage_stat`.

When a non-existent account needs to be represented, the `account_none` constructor is used.

4.1.7. Account state as a message from an account to its future self. Notice that the account state is very similar to a message sent from an account to its future self participating in the next transaction, for the following reasons:

- The account state does not change between two consecutive transactions of the same account, so it is completely similar in this respect to a message sent from the earlier transaction to the later one.
- When a transaction is processed, its inputs are an inbound message and the previous account state; its outputs are outbound messages generated and the next account state. If we treat the state as a special kind of message, we see that every transaction has exactly two inputs (the account state and an inbound message) and at least one output.

- Both a message and the account state can carry code and data in an instance of *StateInit*, and some value in their `balance`.
- An account is initialized by a *constructor message*, which essentially carries the future state and balance of the account.
- On some occasions messages are converted into account states, and vice versa. For instance, when a shardchain merge event occurs, and two accounts that are instances of the same large contract need to be merged, one of them is converted into a message sent to the other one (cf. **4.2.11**). Similarly, when a shardchain split event occurs, and an instance of a large smart contract needs to be split into two, this is achieved by a special transaction that creates the new instance by means of a constructor message sent from the previously existing instance to the new one (cf. **4.2.10**).
- One may say that a message is involved in transferring some information *across space* (between different shardchains, or at least accountchains), while an account state transfers information *across time* (from the past to the future of the same account).

4.1.8. Differences between messages and account states. Of course, there are important differences, too. For example:

- The account state is transferred only "in time" (for a shardchain block to its successor), but never "in space" (from one shardchain to another). As a consequence, this transfer is done implicitly, without creating complete copies of the account state anywhere in the blockchain.
- Storage payments collected by the validators for keeping the account state usually are considerably smaller than message forwarding fees for the same amount of data.
- When an inbound message is delivered to an account, it is the code from the account that is invoked, not the code from the message.

4.1.9. The combined state of all accounts in a shard. The split part of the shardchain state (cf. **1.2.1** and **1.2.2**) is given by

```
_ (HashmapAugE 256 Account CurrencyCollection) = ShardAccounts;
```

This is simply a dictionary with 256-bit *account_id*s as keys and corresponding account states as values, sum-augmented by the balances of the accounts. In this way the sum of balances of all accounts in a shardchain is computed, so that one can easily check the total amount of cryptocurrency "stored" in a shard.

Internal consistency conditions ensure that the address of an account referred to by key k in *SmartAccounts* is indeed equal to k. An additional internal consistency condition requires that all keys k begin with the shard prefix s.

4.1.10. Account owner and interface descriptions. One may want to include some optional information in a controlled account. For example, an individual user or a company may want to add a text description field to their wallet account, with the user's or company's name or address (or their hash, if the information should not be made publicly available). Alternatively, a smart contract may offer a machine-readable or human-readable description of its supported methods and their intended application, which might be used by advanced wallet applications to construct drop-down menus and forms helping a human user to create valid messages to be sent to that smart contract.

One way of including such information is to reserve, say, the second reference in the `data` cell of the state of an account for a dictionary with 64-bit keys (corresponding to some identifiers of the standard types of extra data one might want to store) and corresponding values. Then a blockchain explorer would be able to extract the required value, along with a Merkle proof if necessary.

A better way of doing this is by defining some *get methods* in the smart contract.

4.1.11. Get methods of a smart contract. *Get methods* are executed by a stand-alone instance of TVM with the account's code and data loaded into it. The required parameters are passed on the stack (say, a magic number indicating the field to be fetched or the specific get method to be invoked), and the results are returned on the TVM stack as well (say, a cell slice containing the serialization of a string with the account owner's name).

As a bonus, get methods may be used to get answers to more sophisticated queries than just fetching a constant object. For instance, TON DNS registry smart contracts provide get methods to look up a domain string in the registry and return the corresponding record, if found.

By convention, get methods use large *negative* 32-bit or 64-bit indices or magic numbers, and internal functions of a smart contract use consecutive *positive* indices, to be used in TVM's `CALLDICT` instruction. The `main()` function of a smart contract, used to process inbound messages in ordinary transactions, always has index zero.

4.2 Transactions

According to the Infinite Sharding Paradigm and the actor model, the three principal components of the TON Blockchain are *accounts* (along with their states), *messages,* and *transactions*. Previous sections have already discussed the first two; this section considers transactions.

In contrast with messages, which have essentially the same headers throughout all workchains of the TON Blockchain, and accounts, which have at least some common parts (the address and the balance), our discussion of transactions is necessarily limited to the masterchain and the basic workchain. Other workchains may define completely different kinds of transactions.

4.2.1. Logical time of a transaction. Each transaction t has a logical time interval $\text{LT}^\bullet(t) = [\text{LT}^-(t), \text{LT}^+(t))$ assigned to it (cf. **1.4.6** and **1.4.3**). By convention, a transaction t generating n outbound messages $m_1, \ldots, m_n$ is assigned a logical time interval of length $n + 1$, so that

$$\text{LT}^+(t) = \text{LT}^-(t) + n + 1 \quad . \tag{16}$$

We also set $\text{LT}(t) := \text{LT}^-(t)$, and assign the logical creation time of message m_i, where $1 \leq i \leq n$, by

$$\text{LT}(m_i) = \text{LT}^-(m_i) := \text{LT}^-(t) + i, \quad \text{LT}^+(m_i) := \text{LT}^-(m_i) + 1 \quad . \tag{17}$$

In this way, each generated outbound message is assigned its own unit interval inside the logical time interval $\text{LT}^\bullet(t)$ of transaction t.

4.2.2. Logical time uniquely identifies transactions and outbound messages of an account. Recall that the conditions imposed on logical time imply that $\text{LT}^-(t) \geq \text{LT}^+(t')$ for any preceding transaction t' of the same account ξ, and that $\text{LT}^-(t) > \text{LT}(m)$ if m is the inbound message processed by transaction t. In this way, the logical time intervals of transactions of the same account do not intersect each other, and as a consequence, the logical time intervals of all outbound messages generated by an account do

not intersect each other either. In other words, all $\text{Lt}(m)$ are different, when m runs through all outbound messages of the same account ξ.

In this way, $\text{Lt}(t)$ and $\text{Lt}(m)$, when combined with an account identifier ξ, uniquely determine a transaction t or an outbound message m of that account. Furthermore, if one has an ordered list of all transactions of an account along with their logical times, it is easy to find the transaction that generated a given outbound message m, simply by looking up the transaction t with logical time $\text{Lt}(t)$ nearest to $\text{Lt}(m)$ from below.

4.2.3. Generic components of a transaction. Each transaction t contains or indirectly refers to the following data:

- The account ξ to which the transaction belongs.
- The logical time $\text{Lt}(t)$ of the transaction.
- One or zero inbound messages m processed by the transaction.
- The number of generated outbound messages $n \geq 0$.
- The outbound messages $m_1, \ldots, m_n$.
- The initial state of account ξ (including its balance).
- The final state of account ξ (including its balance).
- The total fees collected by the validators.
- A detailed description of the transaction containing all or some data needed to validate it, including the kind of the transaction (cf. **4.2.4**) and some of the intermediate steps performed.

Of these components, all but the very last one are quite general and might appear in other workchains as well.

4.2.4. Kinds of transactions. There are different kinds of transactions allowed in the masterchain and the shardchains. *Ordinary* transactions consist in the delivery of one inbound message to an account, and its processing by that account's code; this is the most common kind of transaction. Additionally, there are several kinds of *exotic* transactions.

Altogether, there are six kinds of transactions:

- *Ordinary transactions* — Belong to an account ξ. They process exactly one inbound message m (described in *InMsgDescr* of the encompassing block) with destination ξ, compute the new state of the account, and generate several outbound messages (registered in *OutMsgDescr*) with source ξ.

- *Storage transactions* — Can be inserted by validators at their discretion. They do not process any inbound message and do not invoke any code. Their only effect is to collect storage payments from an account, affecting its storage statistics and its balance. If the resulting Gram balance of the account becomes less than a certain amount, the account may be frozen and its code and data replaced by their combined hash.

- *Tick transactions* — Automatically invoked for certain special accounts (smart contracts) in the masterchain that have the `tick` flag set in their state, as the very first transactions in every masterchain block. They have no inbound message, but may generate outbound messages and change the account state. For instance, *validator elections* are performed by tick transactions of special smart contracts in the masterchain.

- *Tock transactions* — Similarly automatically invoked as the very last transactions in every masterchain block for certain special accounts.

- *Split transactions* — Invoked as the last transactions of shardchain blocks immediately preceding a shardchain split event. They are triggered automatically for instances of large smart contracts that need to produce a new instance after splitting.

- *Merge transactions* — Similarly invoked as the first transactions of shardchain blocks immediately after a shardchain merge event, if an instance of a large smart contract needs to be merged with another instance of the same smart contract.

Notice that out of these six kinds of transactions, only four can occur in the masterchain, and another subset of four can occur in the basic workchain.

4.2.5. Phases of an ordinary transaction. An ordinary transaction is performed in several *phases*, which may be thought of as several "subtransactions" tightly bound into one:

- *Storage phase* — Collects due storage payments for the account state (including smart-contract code and data, if present) up to the present time. The smart contract may be *frozen* as a result. If the smart contract did not exist before, the storage phase is absent.

- *Credit phase* — The account is credited with the value of the inbound message received.

- *Computing phase* — The code of the smart contract is invoked inside an instance of TVM with adequate parameters, including a copy of the inbound message and of the persistent data, and terminates with an exit code, the new persistent data, and an *action list* (which includes, for instance, outbound messages to be sent). The processing phase may lead to the creation of a new account (uninitialized or active), or to the activation of a previously uninitialized or frozen account. The *gas payment*, equal to the product of the gas price and the gas consumed, is exacted from the account balance.

- *Action phase* — If the smart contract has terminated successfully (with exit code 0 or 1), the actions from the list are performed. If it is impossible to perform all of them—for example, because of insufficient funds to transfer with an outbound message—then the transaction is aborted and the account state is rolled back. The transaction is also aborted if the smart contract did not terminate successfully, or if it was not possible to invoke the smart contract at all because it is uninitialized or frozen.

- *Bounce phase* — If the transaction has been aborted, and the inbound message has its `bounce` flag set, then it is "bounced" by automatically generating an outbound message (with the `bounce` flag clear) to its original sender. Almost all value of the original inbound message (minus gas payments and forwarding fees) is transferred to the generated message, which otherwise has an empty body.

4.2.6. Bouncing inbound messages to non-existent accounts. Notice that if an inbound message with its `bounce` flag set is sent to a previously non-existent account, and the transaction is aborted (for instance, because there is no code and data with the correct hash in the inbound message, so the virtual machine could not be invoked at all), then the account is not

created even as an uninitialized account, since it would have zero balance and no code and data anyways.[31]

4.2.7. Processing of an inbound message is split between computing and action phases. Notice that the processing of an inbound message is in fact split into two phases: the *computing* phase and the *action* phase. During the computing phase, the virtual machine is invoked and the necessary computations are performed, but no actions outside the virtual machine are taken. In other words, *the execution of a smart contract in TVM has no side effects*; there is no way for a smart contract to interact with the blockchain directly during its execution. Instead, TVM primitives such as `SENDMSG` simply store the required action (e.g., the outbound message to be sent) into the action list being gradually accumulated in TVM control register `c6`. The actions themselves are postponed until the action phase, during which the user smart contract is not invoked at all.

4.2.8. Reasons for splitting the processing into computation and action phases. Some reasons for such an arrangement are:

- It is simpler to abort the transaction if the smart contract eventually terminates with an exit code other than 0 or 1.
- The rules for processing output actions may be changed without modifying the virtual machine. (For instance, new output actions may be introduced.)
- The virtual machine itself may be modified or even replaced by another one (for instance, in a new workchain) without changing the rules for processing output actions.
- The execution of the smart contract inside the virtual machine is completely isolated from the blockchain and is a *pure computation*. As a consequence, this execution may be *virtualized* inside the virtual machine itself by means of TVM's `RUNVM` primitive, a useful feature for validator smart contracts and for smart contracts controlling payment

[31] In particular, if a user mistakenly sends some funds to a non-existent address in a bounceable message, the funds will not be wasted, but rather will be returned (bounced) back. Therefore, a user wallet application should set the `bounce` flag in all generated messages by default unless explicitly instructed otherwise. However, non-bounceable messages are indispensable in some situations (cf. **1.7.6**).

channels and other sidechains. Additionally, the virtual machine may be *emulated* inside itself or a stripped-down version of itself, a useful feature for validating the execution of smart contracts inside TVM.[32]

4.2.9. Storage, tick, and tock transactions. Storage transactions are very similar to a stand-alone storage phase of an ordinary transaction. Tick and tock transactions are similar to ordinary transactions without credit and bounce phases, because there is no inbound message.

4.2.10. Split transactions. Split transactions in fact consist of two transactions. If an account ξ needs to be split into two accounts ξ and ξ':

- First a *split prepare transaction*, similar to a tock transaction (but in a shardchain instead of the masterchain), is issued for account ξ. It must be the last transaction for ξ in a shardchain block. The output of the processing stage of a split prepare transaction consists not only of the new state of account ξ, but also of the new state of account ξ', represented by a constructor message to ξ' (cf. **4.1.7**).
- Then a *split install transaction* is added for account ξ', with a reference to the corresponding split prepare transaction. The split install transaction must be the only transaction for a previously non-existent account ξ' in the block. It effectively sets the state of ξ' as defined by the split prepare transaction.

4.2.11. Merge transactions. Merge transactions also consist of two transactions each. If an account ξ' needs to be merged into account ξ:

- First a *merge prepare transaction* is issued for ξ', which converts all of its persistent state and balance into a special constructor message with destination ξ (cf. **4.1.7**).
- Then a *merge install transaction* for ξ, referring to the corresponding merge prepare transaction, processes that constructor message. The merge install transaction is similar to a tick transaction in that it must be the first transaction for ξ in a block, but it is located in a shardchain block, not in the masterchain, and it has a special inbound message.

[32] A reference implementation of a TVM emulator running in a stripped-down version of TVM may be committed into the masterchain to be used when a disagreement between the validators on a specific run of TVM arises. In this way, flawed implementations of TVM may be detected. The reference implementation then serves as an authoritative source on the operational semantics of TVM. (Cf. [4, B.2])

4.2.12. Serialization of a general transaction. Any transaction contains the fields listed in **4.2.3**. As a consequence, there are some common components in all transactions:

```
transaction$_ account_addr:uint256 lt:uint64 outmsg_cnt:uint15
  orig_status:AccountStatus end_status:AccountStatus
  in_msg:(Maybe ^Message) out_msgs:(HashmapE 15 ^Message)
  total_fees:Grams state_update:^(MERKLE_UPDATE Account)
  description:^TransactionDescr = Transaction;

!merkle_update#02 {X:Type} old_hash:uint256 new_hash:uint256
  old:^X new:^X = MERKLE_UPDATE X;
```

The exclamation mark in the TL-B declaration of a `merkle_update` indicates special processing required for such values. In particular, they must be kept in a separate cell, which must be marked as *exotic* by a bit in its header (cf. [4, 3.1]).

A full explanation of the serialization of *TransactionDescr*, which describes one transaction according to its kind listed in **4.2.4**, can be found in **4.3**.

4.2.13. Representation of outbound messages generated by a transaction. The outbound messages generated by a transaction t are kept in a dictionary `out_msgs` with 15-bit keys equal to 0, 1, ..., $n-1$, where $n =$ `outmsg_cnt` is the number of generated outbound messages. Message m_{i+1} with index $0 \leq i < n$ must have $\mathrm{LT}(m_{i+1}) = \mathrm{LT}(t) + i + 1$, and $\mathrm{LT}(t) = \mathrm{LT}^{-}(t)$ is explicitly stored in the `lt` field.

4.2.14. Consistency conditions for transactions. The common serialization of the fields present in a *Transaction*, independent of its type and description, enables us to impose several "external" consistency conditions on any transaction. The most important of them involves the *value flow* inside the transaction: the sum of all inputs (the import value of the inbound message plus the original balance of the account) must equal the sum of all outputs (the resulting balance of the account, plus the sum of the export values of all outbound messages, plus all storage, processing, and forwarding fees collected by the validators). In this way, a surface inspection of a transaction, which processes an inbound message with an import value of 1 Gram received by an account with an initial balance of 10 Grams, generating an

outbound message with an export value of 100 Grams in the process, will reveal its invalidity even before checking all the details of the TVM execution.

Other consistency conditions may slightly depend on the description of the transaction. For instance, the inbound message processed by an ordinary transaction must be registered in the *InMsgDescr* of the encompassing block, and the corresponding record must contain a reference to this transaction. Similarly, all outbound messages generated by all transactions (with the exception of one special message generated by a split prepare or merge prepare transaction) must be registered in *OutMsgDescr*.

4.2.15. Collection of all transactions of an account. All transactions in a block belonging to the same account ξ are collected into an "accountchain block" *AccountBlock,* which essentially is a dictionary `transactions` with 64-bit keys, each equal to the logical time of the corresponding transaction:

```
acc_trans$_ account_addr:uint256
            transactions:(HashmapAug 64 ^Transaction Grams)
            state_update:^(MERKLE_UPDATE Account)
          = AccountBlock;
```

The `transactions` dictionary is sum-augmented by a *Grams* value, which aggregates the total fees collected from these transactions.

In addition to this dictionary, an *AccountBlock* contains a Merkle update (cf. [4, 3.1]) of the total state of the account. If an account did not exist before the block, its state is represented by an `account_none`.

4.2.16. Consistency conditions for *AccountBlock*s. There are several general consistency conditions imposed on an *AccountBlock*. In particular:

- The transaction appearing as a value in the augmented `transactions` dictionary must have its `lt` value equal to its key.
- All transactions must belong to an account whose address `account_addr` is indicated in the *AccountBlock*.
- If t and t' are two transactions with $\text{Lt}(t) < \text{Lt}(t')$, and their keys are consecutive in `transactions`, meaning that there is no transaction t'' with $\text{Lt}(t) < \text{Lt}(t'') < \text{Lt}(t')$, then the final state of t must correspond to the initial state of t' (their hashes as explicitly indicated in the Merkle updates must be equal).

- If t is the transaction with minimal $\text{Lt}(t)$, its initial state must coincide with the initial state as indicated in `state_update` of the *AccountBlock*.
- If t is the transaction with maximal $\text{Lt}(t)$, its final state must coincide with the final state as indicated in `state_update` of the *AccountBlock*.
- The list of transactions must be non-empty.

These conditions simply express the fact that the state of an account may change only as the result of performing a transaction.

4.2.17. Collection of all transactions in a block. All transactions in a block are represented by (cf. **1.2.1**):

```
_ (HashmapAugE 256 AccountBlock Grams) = ShardAccountBlocks;
```

4.2.18. Consistency conditions for the collection of all transactions. Again, consistency conditions are imposed on this structure, requiring that the value at key ξ be an *AccountBlock* with address equal to ξ. Further consistency conditions relate this structure with the initial and final states of the shardchain indicated in the block, requiring that:

- If *ShardAccountBlock* has no key ξ, then the state of account ξ in the initial and in the final state of the block must coincide (or it must be absent from both).
- If ξ is present in *ShardAccountBlock*, its initial and final states as indicated in *AccountBlock* must match those indicated in the initial and final states of the shardchain block, expressed by instances of *ShardAccounts* (cf. **4.1.9**).

These conditions express that the shardchain state is indeed composed out of the states of separate accountchains.

4.3 Transaction descriptions

This section presents the specific TL-B schemes for transaction descriptions according to the classification provided in **4.2.4**.

4.3.1. Reasons for omitting data from a transaction description. A transaction description for a blockchain featuring a Turing-complete virtual machine for smart-contract execution is necessarily incomplete. Indeed, a truly complete description would contain all the intermediate states of the virtual machine after each instruction is executed, something that cannot fit into a blockchain block of a reasonable size. Therefore, the description of such a transaction is likely to contain only the total number of steps and the hashes of the initial and final states of the virtual machine. The validation of such a transaction will necessarily involve the execution of the smart contract to reproduce all the intermediate steps and the final result.

If we compress the sequence of all intermediate steps of the virtual machine into just the hashes of the initial and final states, then no transaction details at all need to be included: a validator able to check the execution of the virtual machine by itself would also be able to check all the other actions of the transaction starting from its initial data without these details.

4.3.2. Reasons for including data into a transaction description. The above considerations notwithstanding, there are still several reasons to introduce some details in the transaction description:

- We want to impose external consistency conditions on the transaction, so that at least the validity of the value flow inside the transaction and the validity of inbound and outbound messages can be quickly checked without invoking the virtual machine (cf. **4.2.14**). This at least guarantees the invariance of the total amount of each cryptocurrency in the blockchain, even if it does not guarantee the correctness of its distribution.

- We want to be able to trace principal state changes of an account (such as its being created, activated, or frozen) by inspecting the data stored in the transaction description, without figuring out the missing details of the transaction. This simplifies the verification of the consistency conditions between the accountchain and shardchain states in a block.

- Finally, certain information—such as the total steps of the virtual machine, the hashes of its initial and final states, the total gas consumed, and the exit code—might considerably simplify the debugging and implementation of the TON Blockchain software. (This information would help a human programmer understand what has happened in a particular blockchain block.)

On the other hand, we want to minimize the size of each transaction, because we want to maximize the number of transactions that can fit into each (bounded-size) block. Therefore, all information not required for one of the above reasons is omitted.

4.3.3. Description of a storage phase. The storage phase is present in several kinds of transactions, so a common representation for this phase is used:

```
tr_phase_storage$_ storage_fees_collected:Grams
  storage_fees_due:(Maybe Grams)
  status_change:AccStatusChange
  = TrStoragePhase;

acst_unchanged$0 = AccStatusChange;  // x -> x
acst_frozen$10 = AccStatusChange;    // init -> frozen
acst_deleted$11 = AccStatusChange;   // frozen -> deleted
```

4.3.4. Description of a credit phase. The credit phase can result in the collection of some due payments:

```
tr_phase_credit$_ due_fees_collected:(Maybe Grams)
  credit:CurrencyCollection = TrCreditPhase;
```

The sum of `due_fees_collected` and `credit` must equal the value of the message received, plus its `ihr_fee` if the message has not been received via IHR (otherwise the `ihr_fee` is awarded to the validators).

4.3.5. Description of a computing phase. The computing phase consists in invoking TVM with correct inputs. On some occasions, TVM cannot be invoked at all (e.g., if the account is absent, not initialized, or frozen, and the inbound message being processed has no code or data fields or these fields have an incorrect hash); this is reflected by corresponding constructors.

```
tr_phase_compute_skipped$0 reason:ComputeSkipReason
  = TrComputePhase;
tr_phase_compute_vm$1 success:Bool msg_state_used:Bool
  account_activated:Bool gas_fees:Gram
  _:^[ gas_used:(VarUInteger 7)
  gas_limit:(VarUInteger 7) gas_credit:(Maybe (VarUInteger 3))
```

```
  mode:int8 exit_code:int32 exit_arg:(Maybe int32)
  vm_steps:uint32
  vm_init_state_hash:uint256 vm_final_state_hash:uint256 ]
  = TrComputePhase;
cskip_no_state$00 = ComputeSkipReason;
cskip_bad_state$01 = ComputeSkipReason;
cskip_no_gas$10 = ComputeSkipReason;
```

The TL-B construct `_:^[...]` describes a reference to a cell containing the fields listed inside the square brackets. In this way, several fields can be moved from a cell containing a large record into a separate subcell.

4.3.6. Skipped computing phase. If the computing phase has been skipped, possible reasons include:

- The absence of funds to buy gas.
- The absence of a state (i.e., smart-contract code and data) in both the account (non-existing, uninitialized, or frozen) and the message.
- An invalid state passed in the message (i.e., the state's hash differs from the expected value) to a frozen or uninitialized account.

4.3.7. Valid computing phase. If there is no reason to skip the computing phase, TVM is invoked and the results of the computation are logged. Possible parameters are as follows:

- The `success` flag is set if and only if `exit_code` is either 0 or 1.
- The `msg_state_used` parameter reflects whether the state passed in the message has been used. If it is set, the `account_activated` flag reflects whether this has resulted in the activation of a previously frozen, uninitialized or non-existent account.
- The `gas_fees` parameter reflects the total gas fees collected by the validators for executing this transaction. It must be equal to the product of `gas_used` and `gas_price` from the current block header.
- The `gas_limit` parameter reflects the gas limit for this instance of TVM. It equals the lesser of either the Grams credited in the credit phase from the value of the inbound message divided by the current gas price, or the global per-transaction gas limit.

- The `gas_credit` parameter may be non-zero only for external inbound messages. It is the lesser of either the amount of gas that can be paid from the account balance or the maximum gas credit.
- The `exit_code` and `exit_args` parameters represent the status values returned by TVM.
- The `vm_init_state_hash` and `vm_final_state_hash` parameters are the representation hashes of the original and resulting states of TVM, and `vm_steps` is the total number of steps performed by TVM (usually equal to two plus the number of instructions executed, including implicit `RET`s).[33]

4.3.8. Description of the action phase. The action phase occurs after a valid computation phase. It attempts to perform the actions stored by TVM during the computing phase into the *action list*. It may fail, because the action list may turn out to be too long, contain invalid actions, or contain actions that cannot be completed (for instance, because of insufficient funds to create an outbound message with the required value).

```
tr_phase_action$_ success:Bool valid:Bool no_funds:Bool
  status_change:AccStatusChange
  total_fwd_fees:(Maybe Grams) total_action_fees:(Maybe Grams)
  result_code:int32 result_arg:(Maybe int32) tot_actions:int16
  spec_actions:int16 msgs_created:int16
  action_list_hash:uint256 tot_msg_size:StorageUsed
  = TrActionPhase;
```

4.3.9. Description of the bounce phase.

```
tr_phase_bounce_negfunds$00 = TrBouncePhase;
tr_phase_bounce_nofunds$01 msg_size:StorageUsed
  req_fwd_fees:Grams = TrBouncePhase;
tr_phase_bounce_ok$1 msg_size:StorageUsed
  msg_fees:Grams fwd_fees:Grams = TrBouncePhase;
```

4.3.10. Description of an ordinary transaction.

[33] Notice that this record does not represent a change in the state of the account, because the transaction may still be aborted during the action phase. In that case, the new persistent data indirectly referenced by `vm_final_state_hash` will be discarded.

```
trans_ord$0000 storage_ph:(Maybe TrStoragePhase)
  credit_ph:(Maybe TrCreditPhase)
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Boolean bounce:(Maybe TrBouncePhase)
  destroyed:Boolean
  = TransactionDescr;
```

Several consistency conditions are imposed on this structure:

- `action` is absent if and only if the computing phase was unsuccessful.
- The `aborted` flag is set either if there is no action phase or if the action phase was unsuccessful.
- The bounce phase occurs only if the `aborted` flag is set and the inbound message was bounceable.

4.3.11. Description of a storage transaction. A storage transaction consists just of one stand-alone storage phase:

```
trans_storage$0001 storage_ph:TrStoragePhase
  = TransactionDescr;
```

4.3.12. Description of tick and tock transactions. Tick and tock transactions are similar to ordinary transactions without an inbound message, so there are no credit or bounce phases:

```
tick$0 = TickTock;
tock$1 = TickTock;
trans_tick_tock$001 tt:TickTock storage:TrStoragePhase
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Boolean destroyed:Boolean = TransactionDescr;
```

4.3.13. Split prepare and install transactions. A split prepare transaction is similar to a tock transaction in a masterchain, but it must generate exactly one special constructor message; otherwise, the action phase is aborted.

```
split_merge_info$_ cur_shard_pfx_len:(## 6)
  acc_split_depth:(##6) this_addr:uint256 sibling_addr:uint256
  = SplitMergeInfo;
```

```
trans_split_prepare$0100 split_info:SplitMergeInfo
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Boolean destroyed:Boolean
  = TransactionDescr;
trans_split_install$0101 split_info:SplitMergeInfo
  prepare_transaction:^Transaction
  installed:Boolean = TransactionDescr;
```

Notice that the split install transaction for the new sibling account ξ' refers to its corresponding split prepare transaction of the previously existing account ξ.

4.3.14. Merge prepare and install transactions. A merge prepare transaction converts the state and balance of an account into a message, and a subsequent merge install transaction processes this state:

```
trans_merge_prepare$0110 split_info:SplitMergeInfo
  storage_ph:TrStoragePhase aborted:Boolean
  = TransactionDescr;
trans_merge_install$0111 split_info:SplitMergeInfo
  prepare_transaction:^Transaction
  credit_ph:(Maybe TrCreditPhase)
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Boolean destroyed:Boolean
  = TransactionDescr;
```

4.4 Invoking smart contracts in TVM

This section describes the exact parameters with which TVM is invoked during the computing phase of ordinary and other transactions.

4.4.1. Smart-contract code. The *code* of a smart contract is normally a part of the account's persistent state, at least if the account is *active* (cf. **4.1.6**). However, a frozen or uninitialized (or non-existent) account has no persistent state, with the possible exception of the account's balance and the hash of its intended state (equal to the account address for uninitialized accounts). In this case, the code must be supplied in the `init` field of the inbound message being processed by the transaction (cf. **3.1.7**).

4.4.2. Smart-contract persistent data. The *persistent data* of a smart contract is kept alongside its code, and remarks similar to those made above in **4.4.1** apply. In this respect, the code and persistent data of a smart contract are just two parts of its persistent state, which differ only in the way they are treated by TVM during smart-contract execution.

4.4.3. Smart-contract library environment. The *library environment* of a smart contract is a hashmap mapping 256-bit cell (representation) hashes into the corresponding cells themselves. When an external cell reference is accessed during the execution of a smart contract, the cell referred to is looked up in the library environment and the external cell reference is transparently replaced by the cell found.

The library environment for an invocation of a smart contract is computed as follows:

1. The global library environment for the workchain in question is taken from the current state of the masterchain.[34]

2. Next, it is augmented by the local library environment of the smart contract, stored in the `library` field of the smart contract's state. Only 256-bit keys equal to the hashes of the corresponding value cells are taken into account. If a key is present in both the global and local library environments, the local environment takes precedence while merging the two library environments.

3. Finally, the message library stored in the `library` field of the `init` field of the inbound message is similarly taken into account. Notice, however, that if the account is frozen or uninitialized, the `library` field of the message is part of the suggested state of the account, and is used instead of the local library environment in the previous step. The message library has lower precedence than both the local and the global library environments.

4.4.4. The initial state of TVM. A new instance of TVM is initialized prior to the execution of a smart contract as follows:

- The original `cc` (current continuation) is initialized using the cell slice created from the cell `code`, containing the code of the smart contract computed as described in **4.4.1**.

[34]The most common way of creating shared libraries for TVM is to publish a reference to the root cell of the library in the masterchain.